

AFRL-IF-RS-TM-2007-6
Final Technical Memorandum
February 2007



LOW OVERHEAD SOFTWARE/HARDWARE MECHANISMS FOR SOFTWARE ASSURANCE AND PRODUCIBILITY

SUNY Binghamton

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Rome Research Site Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

**AFRL-IF-RS-TM-2007-6 HAS BEEN REVIEWED AND IS APPROVED FOR
PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION
STATEMENT.**

FOR THE DIRECTOR:

/s/

ANNA L. LEMAIRE
Work Unit Manager

/s/

IGOR G. PLONISCH, Chief
Strategic Planning & Business Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) FEB 2007	2. REPORT TYPE Final	3. DATES COVERED (From - To) May 06 – Sep 06		
4. TITLE AND SUBTITLE LOW OVERHEAD SOFTWARE/HARDWARE MECHANISMS FOR SOFTWARE ASSURANCE AND PRODUCIBILITY		5a. CONTRACT NUMBER		
		5b. GRANT NUMBER FA8750-06-1-0247		
		5c. PROGRAM ELEMENT NUMBER 62702F		
6. AUTHOR(S) Aneesh Aggarwal		5d. PROJECT NUMBER 558B		
		5e. TASK NUMBER II		
		5f. WORK UNIT NUMBER RS		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Aneesh Aggarwal SUNY Binghamton Department of Electrical and Computer Engineering Binghamton NY 13903		8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/IFB 26 Electronic Parkway Rome NY 13441-4514		10. SPONSOR/MONITOR'S ACRONYM(S)		
		11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TM-2007-6		
12. DISTRIBUTION AVAILABILITY STATEMENT <i>APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 07-073</i>				
13. SUPPLEMENTARY NOTES				
14. ABSTRACT Memory related software vulnerabilities such as buffer overflow and dangling pointers make computer systems vulnerable to exploits and cost the US economy huge sums of money. Software tools proposed so far to address these vulnerabilities are limited in their applicability because they either have low detection rate and high false alarm rate or have a huge performance overhead. This report explores a new initiative to develop low overhead integrated hardware/software mechanisms to detect memory related vulnerabilities. These mechanisms are expected to resolve the limitations of software approaches by using specialized hardware for detecting the vulnerabilities, thus tremendously facilitating software assurance and producibility.				
15. SUBJECT TERMS Software vulnerabilities, buffer overflow, low overhead integrated software/hardware, software assurance.				
16. SECURITY CLASSIFICATION OF:		17. LIMITATION OF ABSTRACT UL	18. NUMBER OF PAGES 8	19a. NAME OF RESPONSIBLE PERSON Anna L. Lemaire
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U		19b. TELEPHONE NUMBER (<i>Include area code</i>)

Low Overhead Software/Hardware Mechanisms for Software Assurance and Producibility

Aneesh Aggarwal
SUNY Binghamton
Binghamton, NY 13903
Email: aneesh@binghamton.edu
Phone: 607-777-2509

Abstract

Memory related software vulnerabilities such as buffer overflow and dangling pointers make computer systems vulnerable to exploits and cost the US economy huge sums of money. Software tools proposed so far to address these vulnerabilities are limited in their applicability because they either have low detection rate and high false alarm rate or have a huge performance overhead. This report explores a new initiative to develop low overhead integrated hardware/software mechanisms to detect memory related vulnerabilities. These mechanisms are expected to resolve the limitations of software approaches by using specialized hardware for detecting the vulnerabilities, thus tremendously facilitating software assurance and producibility.

Introduction

Software vulnerabilities make computer systems vulnerable to attacks and exploits. They also result in delays in software deployment and in high rates of computer system failures. Software vulnerabilities account for 40% of computer system failures¹ and cost the U.S. economy an estimated \$59.5 billion². Most of the software vulnerabilities are memory related. For instance, over 60% of CERT (Computer Emergency Response Team <http://www.cert.org>) advisories deal with memory related vulnerabilities³.

The two main types of memory related vulnerabilities are buffer overflows and dangling pointers. Buffer overflow occurs when an access to a buffer is made outside the buffer boundary. For instance, an access to element P[100] of a buffer P with 100 elements lies outside the boundaries of P. Dangling pointers are pointers that point to invalid memory objects. For instance, a pointer to an object that has been deallocated is a dangling pointer. Such pointers become more ominous when the space of the deallocated object is reused by another object, which is then accessed by the dangling pointer. These vulnerabilities are particular difficult to detect because they usually manifest at locations different from the point of origin. Furthermore, these vulnerabilities are among the easiest to exploit for hijacking the computer system and executing malicious code.

This project is a new initiative in developing low overhead (in terms of performance and energy) mechanisms to detect memory related violations. The advantages of such mechanisms are:

1. **Software Assurance:** These mechanisms benefit software assurance in two major ways. Firstly, they enable computer systems to ward off attacks and exploits. For instance, if malicious data is provided to an application running on this system with an intention to overflow the buffer provided for the data, then the system detects and prevents this from happening. This prevents anyone from exploiting the buffer overflow vulnerability to gain control of the system. Secondly, these mechanisms enable users to validate any software downloaded from the internet. The downloaded software is executed on this system to detect and correct memory related vulnerabilities that may be present in the software.
2. **Software Producibility:** These mechanisms also facilitate faster production of software. Memory related errors are difficult to detect and usually result in significant delays in software deliveries. With these mechanisms in place, applications can be periodically executed, during the production phase, on such a system for early detection of errors. Since the system reports the exact locations and types of errors, debugging and hence, production of software becomes much easier and quicker.

The specific goal of this project is to explore low overhead integrated software/hardware mechanisms for software assurance and producibility. In this report, I present the mechanisms and the initial experimental results obtained for the mechanisms. The main goal of the initial experiments is to test the feasibility of the proposed mechanisms.

Technical background

Previous approaches^{4,5,6} have attempted to address the memory related vulnerabilities issues using software tools. These tools fall into two categories: static and dynamic. Static software tools review the application code statically to detect violations. The major limitations of such tools are that they result in a significantly large number of false alarms and that they fail to detect most of the vulnerabilities. Dynamic tools, on the other hand, detect most memory violations and raise an alarm only in the event of a violation. These tools instrument the application code so that when the code executes, the additional instrumentation code checks for memory related violations. A major limitation of such tools is that they execute a significantly large number of instructions in order to detect the violations. This process slows down the execution of the application by an order of magnitude. Hence, these tools cannot be used for software assurance when the application is actually deployed in the field. Furthermore, the performance cost of these tools is high enough to discourage their use (by elongating the detection process) even during the software production phase.

Low Overhead Mechanisms

This project proposes a new integrated software/hardware approach for detection of memory related software vulnerabilities. In this approach, architectural hardware support is provided to expedite the operations involved in the detection of such violations. This hardware is controlled by inserting few specialized instructions in the code. These instructions are inserted during the compilation phase and require no user input. This approach is expected to have low performance and energy overhead due to the use of specialized hardware. However, in order to develop the proposed mechanisms, basic and applied research is required in various areas such as hardware structures for violation detection, software and hardware support and integration, format of the specialized instructions, compilation techniques, etc. Extensive research efforts are also required in optimizing the mechanisms.

The approach taken for detecting buffer overflow vulnerabilities is to use two properties of ANSI C: (1) every pointer value at run-time is derived from the address of a unique object and must only be used to access that object and (2) any arithmetic on a pointer value must ensure that the source and result pointers point to the same object. In this work, we focus on programs written in the C programming language. This does not limit the benefits of the mechanisms, as many applications (especially those that require high performance and cannot incur the overhead of abstraction) are (and will be in the future) written in C. The basic idea is that if the boundaries of a memory object are known, then it is ensured that any access originating from between these bounds remains within the bounds after any arithmetic on the pointer. The discussed implementation records the boundaries of memory objects and the pointer-object associations for pointers during execution of applications. Special instructions are inserted by the compiler (during compilation) at places in the code where pointers and objects are declared, initialized, and defined. For instance, instructions to record the information for the global memory objects and pointers are inserted at the start of the application code, and those to record (and delete) the information for the local pointers and memory objects are inserted at the start (and end) of each procedure. For instance, consider the example shown in Figure 1.

In the example of Figure 1, a local integer pointer (ptr) is declared in the procedure foo. An instruction is inserted, in the compiled code, at the declaration point of ptr to register the pointer. Subsequently, when the pointer ptr is allocated memory location, the pointer-object association is recorded by inserting instructions where malloc is called. The memory object boundary is defined by its starting (returned by malloc) and its ending (computed using the object size) address. When ptr is used by accessing memory location (supposedly inside the associated memory object) at an offset from the ptr, instructions are inserted to ensure that the

accessed memory location is within the memory boundary locations. Eventually when the execution returns from the procedure foo, instructions are inserted to deregister the pointer ptr.

```
void foo()
{
    int ptr; /*insert instruction to record address of ptr*/
    ...
    ptr = malloc(sizeof(int), 4); /* insert instruction to associate ptr with a
    memory object of size 4 integers and have a starting address return by malloc*/
    ...
    ptr[x] = ...; /* insert instruction to check the access using ptr*/
    ...
    return; /* insert instruction to deregister the pointer ptr;
}
```

Figure 1: An example illustrating the insertion of special instructions to ensure correct memory accesses

The architectural support includes context addressable memory (CAM) buffers for associative searches in the metadata stored for pointers and objects. When a pointer is declared (ptr in Figure 1), its address is stored in the buffer and when it is initialized or redefined, the corresponding pointer-object association is updated for the pointer. A CAM structure is used to speedup the associative search for the memory object associated with a pointer. To accommodate large number of pointers that may be present in an application, memory in the application address space is used to back the buffers. Performance impact is limited by keeping the required pointers and objects in the buffers. When pointer addresses are loaded with the intention of accessing data using those pointers (the statement ptr[x] = ... in Figure 1), boundaries of the objects associated with those pointer addresses are recorded. Any arithmetic on pointer addresses (contents of pointer ptr plus the offset x, in Figure 1) carries the associated object boundaries along with the results. When the pointers are eventually used to access memory, the associated object boundaries are checked for boundary violations.

The discussion so far assumes that the source code of the applications is available for compilation. This is a requirement for the proposed system so that the compiler is able to insert the specialized instructions to control the additional hardware for detecting memory related violations. Even with this limitation, the proposed system will be extremely beneficial in ensuring software assurance in most cases as discussed in the introduction section. However, if the source code is not available (*e.g.* software binaries that are sold by companies or downloaded from the internet), then some kind of recompilation may be required to insert the instructions for detecting the violations. This requires extensive research into efficient mechanisms to gather the required information from the executable, *e.g.* determine pointer addresses and memory object boundaries, and insert the appropriate instructions. This research is beyond the scope of this report.

Experimental Setup and Initial Results

To perform the feasibility study, we use the olden benchmarks. These benchmarks are kernel benchmarks that have a large number of pointer operations. We insert assembly instructions at the appropriate locations in the C

code for the benchmarks. We then compile the benchmarks using a cross-compiler for the simulator, and execute the benchmarks on the simulator. For simulations, we use a cycle accurate microarchitectural simulator SimpleScalar that simulates the various hardware resources in a microprocessor. We modify the simulator to include the additional hardware required for executing the special instructions.

Figures 2 and 3 show the miss rates of the accesses to the CAM buffer for the health benchmark. Note that the registration of the pointers is not considered a miss because the pointer is not expected to be present in the buffer. When registering a new pointer in the buffer, we use a least recently used (LRU) policy. Figures 2 and 3 show the results for various fully associative buffer sizes. The figures show that, even with a small buffer of size 64 elements, the miss rate saturates to about 90%. Our initial experiments show that for the health benchmark, the number of additional instructions executed to perform the checks for software assurance and producibility are about 30%, giving a performance degradation of only about 13%.

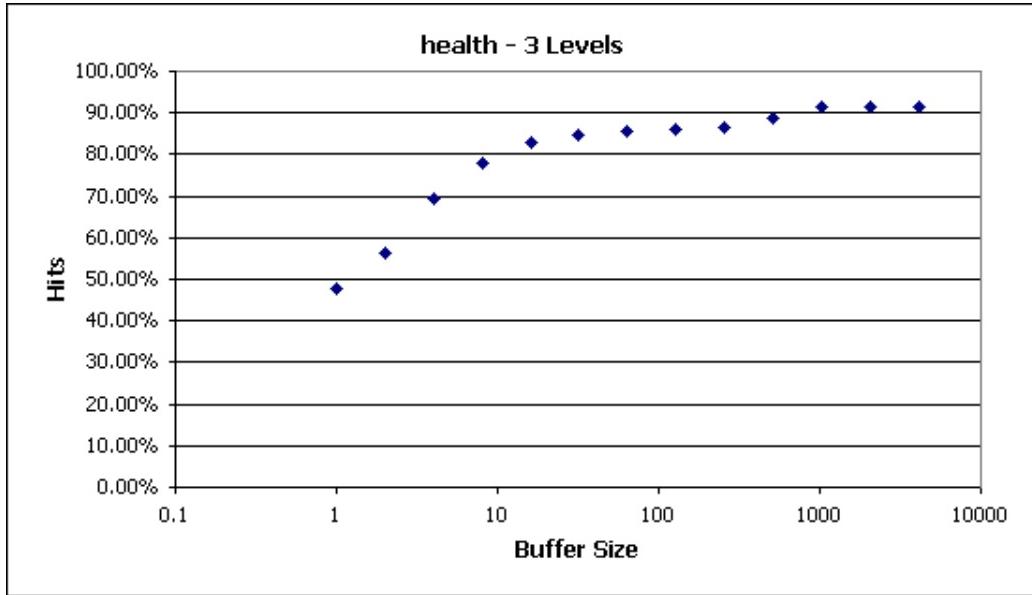


Figure 2: Hit rate in the CAM buffer for health benchmark

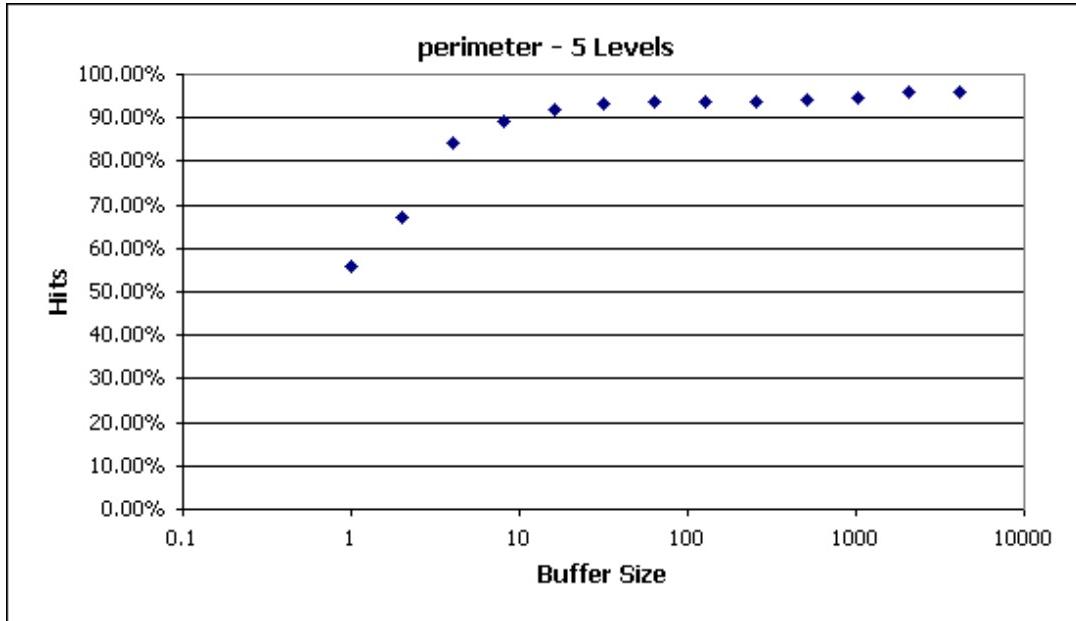


Figure 3: Hit rate in the CAM buffer for perimeter benchmark

Conclusions

Memory related software vulnerabilities such as buffer overflow and dangling pointers make computer systems vulnerable to exploits and cost the US economy huge sums of money. These vulnerabilities are difficult to catch because they manifest at locations that are different from the ones where they occur. Software tools proposed so far to address these vulnerabilities are limited in their applicability because they either have low detection rate and high false alarm rate or have a huge performance overhead. This report performs a feasibility study for a new initiative to develop low overhead integrated hardware/software mechanisms to detect memory related vulnerabilities. In this mechanism, the compiler inserts instructions to make the hardware aware of the declaration, definition, use, and deletion of pointers, along with pointer-object associations. The hardware performs the check at runtime to detect and report memory related violations. Our initial experiments show that even with a large number of pointers and objects, the miss rate in the CAM buffer is reasonably low. Furthermore, the large number of instructions inserted in the code also do not impact performance significantly.

¹ E. Marcus and H. Stern, “Blueprints for high availability,” John Willey and Sons, 2000.

² National Institute of Standards and Technology (NIST), Department of Commerce, “Software errors cost U.S. economy \$59.5 billion annually,” NIST News Release 2002-10, June 2002.

³ www.mcafee.com/us/local_content/white_papers/wp_ricochetbriefbuffer.pdf

⁴ R. Jones and P. Kelly, “Backwards-compatible bounds checking for arrays and pointers in c programs,” Automated and Algorithmic Debugging, 1997.

⁵ O. Ruwase and M. Lam, “A practical dynamic buffer overflow detector,” Network and Distributed System Security (NDSS) Symposium, 2004.

⁶ D. Dhurjati and V. Adve, “Efficiently detecting all dangling pointer uses in production servers,” Dependable Systems and Networks (DSN), 2006.